Comparación de los lenguajes de Programación C y C++ como Herramienta Docente

Vicente Benjumea Manuel Roldán

Resumen

Este documento presenta una comparación entre las primitivas proporcionadas por los lenguajes *C* y *C++*, utilizadas como soporte para la docencia de los conceptos fundamentales de la programación. Nuestra intención no es hacer una comparativa general entre ambos lenguajes, sino únicamente repasar aquellos elementos comunes que pueden ser utilizados en un curso básico de programación. Como resultado del estudio, pensamos que, en general, cuando el objetivo de un curso no es el estudio del lenguaje C, es más adecuado plantear un primer curso de programación sobre la base de C++ que sobre C.

El estudio de *arrays* siempre es parte importante de cualquier primer curso inicial de programación. Tradicionalmente este aspecto ha sido tratado (tanto en *C* como en *C++*) mediante arrays predefinidos. Sin embargo, desde el estándard C++11 (previamente en *TR1*), *C++* dispone de un tipo `array` que pensamos que aporta numerosas ventajas docentes sobre el uso tradicional de arrays predefinidos. Discutimos las características de cada uno y analizamos ventajas e inconvenientes de cada enfoque.

Nota: en todo el texto, salvo que se especifique lo contrario, se hace referencia a *ANSI-C (C89)*.

Índice

1. Definición de Constantes **2**

2. Definición de Variables **3**

3. Control de Ejecución **4**

4. Conversiones de Tipos (castings) **5**

5. Entrada y Salida de Datos **6**

6. Definición de Tipos **11**

7. Subprogramas: Paso de Parámetros **19**

8. El Tipo Puntero y la Gestión de Memoria Dinámica **27**

9. Conclusión **34**

A. Problemas del uso de `printf` y `scanf`: un ejemplo **35**

Este obra está bajo una licencia **Reconocimiento-NoComercial-CompartirIgual 3.0 Unported** de Creative Commons: No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original. Para ver una copia de esta licencia, visite `http://creativecommons.org/licenses/by-nc-sa/3.0/deed.es_ES` o envie una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

1. Definición de Constantes

En el ámbito de la enzeñanza de la programación, es conveniente transmitir la importancia del uso de constantes simbólicas en los programas, evitando en la medida de los posible el uso de constantes literales. A continuación analizamos las posibilidades que ofrecen C y C++ para ello.

1.1. Definición de constantes en C utilizando el preprocesador

En C se pueden definir constantes mediante directivas para el preprocesador. Como consecuencia, puede ocurrir que el valor asociado al símbolo a expandir dependa del contexto en el que expande. Por ejemplo, consideremos las siguientes constantes:

```
#define PI          3.1415
#define CONSTANTE1  10
#define CONSTANTE2  20
#define CONSTANTE3  CONSTANTE1 + CONSTANTE2
```

En este caso la interpretación del símbolo `CONSTANTE3` no es única y depende del contexto en que se expanda dentro del programa. Por ejemplo, supongamos que en la siguiente sentencia queremos almacenar en la variable `v` la mitad del valor asociado al símbolo `CONSTANTE3`:

```
int v = CONSTANTE3 / 2 ;
```

Aunque aparentemente la expresión utilizada para ello es correcta, en realidad es errónea, ya que una vez que el procesador expande el símbolo, la precedencia de los operadores hace que el resultado no sea el previsto y en realidad la variable `v` almacena el valor 20, resultante de evaluar la expresión `10 + 20 / 2`. El mecanismo utilizado para definir las constantes nos ha inducido a un error difícil de detectar, ya que nos encontramos con una expresión final que en realidad está formada por diferentes partes especificadas en puntos diferentes del programa. Obviamente, este problema se puede solucionar incluyendo paréntesis en la definición de la constante.

```
#define CONSTANTE3  (CONSTANTE1 + CONSTANTE2)
```

Sin embargo, debemos ser conscientes de que la utilización del preprocesador para definir constantes simbólicas conlleva la posibilidad de introducir errores inesperados y difíciles de detectar, especialmente en un contexto de aprendizaje de la programación.

1.2. Definición de constantes en C utilizando enumeraciones

También es posible definir constantes de tipo entero (`int`) utilizando enumeraciones:

```
enum {
    CONSTANTE1 = 10,
    CONSTANTE2 = 20,
    CONSTANTE3 = CONSTANTE1 + CONSTANTE2
} ;
```

y utilizar dichas constantes con los valores adecuados.

```
int v = CONSTANTE3 / 2 ;
```

Sin embargo, este mecanismo sólo es adecuado para definir constantes de tipo entero (`int`), no pudiendo ser utilizada para definir constantes de otros tipos, especialmente de coma flotante.

1.3. Definición de constantes en C utilizando `const`

Finalmente, es posible definir *variables inmutables* utilizando la palabra reservada `const`.

```
const int CONSTANTE1 = 10 ;
const int CONSTANTE2 = 20 ;
const int CONSTANTE3 = CONSTANTE1 + CONSTANTE2 ;

int v = CONSTANTE3 / 2 ;
```

Aunque son constantes en el sentido de que no es posible alterar su valor durante la ejecución del programa, no son constantes desde el punto de vista del compilador y por lo tanto no pueden ser utilizadas para aquellas circunstancias donde sea necesaria una constante simbólica definida en *tiempo de compilación* (por ejemplo no son válidas para definir tamaños de arrays). Lo siguiente sería incorrecto:

```
const int TAM = 10 ;
int v[TAM] ;
```

1.4. Definición de constantes en C++

En C++ es posible definir constantes simbólicas de diversos tipos utilizando la palabra reservada `const`. Los valores de estas constantes simbólicas son conocidos por el compilador en *Tiempo de Compilación* y pueden ser utilizadas en cualquier ámbito en el que sean necesarias.

```
const double PI = 3.1415 ;

const int CONSTANTE1 = 10 ;
const int CONSTANTE2 = 20 ;
const int CONSTANTE3 = CONSTANTE1 + CONSTANTE2 ;

int v = CONSTANTE3 / 2 ;
```

2. Definición de Variables

2.1. Definición de variables en C

En C, las variables se deben definir en una zona al comienzo de cada *bloque*, y no pueden mezclarse con las sentencias ejecutables.[1] Así mismo, también es posible inicializar las variables en el lugar en que son definidas.

```
int main()
{
    int x, y ;
    int z = 5 ;

    scanf(" %d", &y) ;
    x = y * z ;
}
```

[1] Sin embargo, en *C99* si es posible entremezclar definición de variables y sentencias ejecutables.

2.2. Definición de variables en C++

En C++, las variables se pueden definir en cualquier parte del *bloque*, pudiendo mezclarse con las sentencias ejecutables, de hecho, se consideran también sentencias ejecutables (invocación al constructor). Esto aumenta la localidad del código, permite declarar las variables en el punto más cercano al lugar de su utilización, y permite declarar e inicializar las variables con los valores adecuados, en vez de realizar una declaración de la variable sin inicializar.

```
int main()
{
    int z = 5 ;
    int y ;
    cin >> y ;
    int x = y * z ;
}
```

3. Control de Ejecución

3.1. Estructuras de control en C y C++

Aunque ambos lenguajes disponen de otras estructuras de control de la ejecución, en el contexto del aprendizaje de programación en un ámbito académico, basado en un enfoque de diseño descendente, abstracción procedimental y programación estructurada, sólo consideraremos las siguientes estructuras de control:

- **Secuencia**: Un bloque determina un secuencia de sentencias ejecutables. Las sentencias se ejecutan secuencialmente.

```
int main()
{
    int x, y = 4 ;
    int z = 5 ;

    x = y * z ;
    z = z + x ;
}
```

- **Selección**: Seleccionan la ejecución de determinados bloques de sentencias, dependiendo del estado de la computación en un momento determinado.

if:

```
int main()
{
    int x = 3, y = 5 ;
    if (x > y * 2) {
        x = y + 2 ;
        y = 0 ;
    } else if (x > y) {
        x = 0 ;
    } else {
        y = 0 ;
        ++x ;
    }
}
```

switch:

```
int main()
{
    int x = 4 ;
    switch (x * 3) {
    case 0:
        x += 2 ;
        ++x ;
        break ;
    case 18:
    case 21:
        --x ;
        break ;
    default:
        x *= 5 ;
        break ;
    }
}
```

- **Iteración**: Repite la ejecución de un bloque de sentencias, dependiendo del estado de la computación en un momento determinado.

for:

```
int main()
{
    int s = 0 ;
    int i ;
    for (i = 0 ; i < 10 ; ++i) {
        s += i ;
    }
}
```

while:

```
int main()
{
    int s = 0 ;
    int i = 0 ;
    while (i < 10) {
        s += i ;
        ++i ;
    }
}
```

do-while:

```
int main()
{
    int s = 0 ;
    int i = 0 ;
    do {
        s += i ;
        ++i ;
    } while (i < 10) ;
}
```

3.2. Características propias de C++

La única diferencia significativa que introduce C++ respecto a C en este aspecto es que permite definir la variable de control del bucle **for** dentro de la propia definición del bucle (en cuyo caso su ámbito de vida se restringe al propio bucle donde está definida), aumentando así la localidad y simplificando la estructura (este mecanismo también ha sido incorporado a *C99*).

```
int main()
{
    int s = 0 ;
    for (int i = 0 ; i < 10 ; ++i) {
        s += i ;
    }
}
```

4. Conversiones de Tipos (castings)

4.1. Conversiones de tipos en C

En C, la conversión de tipos se realiza mediante el operador prefijo de *casting*, donde se especifica entre paréntesis el tipo destino al que se quiere convertir una determinada expresión, seguida por la expresión cuyo resultado quiere ser convertida. Por ejemplo:

```
int main()
{
    int x = (int)3.1415 ;
}
```

Hay que tener presente que la precedencia del operador de casting es la segunda más alta (por debajo del operador punto e indexación), por lo que resulta fácil introducir errores inesperados. Por ejemplo:

```
int main()
{
    double pi = 3.1415 ;
    int x = (int)pi+0.5 ;
}
```

en lugar de:

```
int main()
{
    double pi = 3.1415 ;
    int x = (int)(pi+0.5) ;
}
```

4.2. Conversiones de tipos en C++

En C++, la conversión de tipos se realiza mediante el operador prefijo de *casting*, donde se especifica el tipo destino al que se quiere convertir una determinada expresión, seguida por la expresión *entre paréntesis* cuyo resultado quiere ser convertida. De esta forma, se evitan los posibles problemas derivados de la precedencia de los operadores. Por ejemplo:

```
int main()
{
    int x = int(3.1415) ;

    double pi = 3.1415 ;
    int z = int(pi+0.5) ;
}
```

En caso de tipos con nombres compuestos (`unsigned short`, `unsigned long`, `long double`, `long long`, `unsigned long long`), se debe utilizar un alias (definido con `typedef`):

```
typedef unsigned long ulong ;
int main()
{
    ulong x = ulong(3.1415) ;

    double pi = 3.1415 ;
    ulong z = ulong(pi+0.5) ;
}
```

o utilizar la conversión al estilo de C:

```
int main()
{
    unsigned long x = (unsigned long)3.1415 ;

    double pi = 3.1415 ;
    unsigned long z = (unsigned long)(pi+0.5) ;
}
```

5. Entrada y Salida de Datos

5.1. Entrada y salida de datos en C

En C, la entrada y salida de datos básica se realiza a través de las funciones `scanf` y `printf`, especificando una cadena de caracteres que define el formato y tipos de los datos involucrados, como en el siguiente ejemplo:

```
#include <stdio.h>
int main()
{
    char c = 'a' ;
    short s = 1 ;
    int x = 2 ;
    unsigned u = 3 ;
    long l = 4 ;
    float f = 5.0 ;
    double d = 6.0 ;
    long double dd = 7.0 ;
    char nombre[30] = "pepe" ;

    printf("%c %hd %d %u %ld %g %g %Lg %s\n",
             c,  s, x, u,  l, f, d, dd, nombre) ;

    scanf("%c %hd %d %u %ld %g %lg %Lg %29s\n",
           &c, &s, &x, &u, &l, &f, &d, &dd, nombre) ;
}
```

5.2. Entrada y salida de datos en C++

En C++, la entrada y salida de datos básica se realiza a través de los flujos de entrada (`cin`) y de salida (`cout`) mediante operadores adecuados para ello:

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
int main()
{
    char c = 'a' ;
    short s = 1 ;
    int x = 2 ;
    unsigned u = 3 ;
    long l = 4 ;
    float f = 5.0 ;
    double d = 6.0 ;
    long double dd = 7.0 ;
    string nombre_1 = "juan" ;
    char nombre_2[30] = "pepe" ;

    cout << c << ' ' << s << ' ' << x << ' ' << u << ' ' << l << ' '
         << f << ' ' << d << ' ' << dd << ' ' << nombre_1 << ' '
         << nombre_2 << endl ;

    cin >> c >> s >> x >> u >> l
        >> f >> d >> dd >> nombre_1 >> setw(30) >> nombre_2 ;
}
```

5.3. Salida formateada de datos en C

En C, la salida de datos formateada está orientada al dato. A continuación se muestra un ejemplo de salida formateda de datos en C, donde el símbolo `#` representa el espacio en blanco:

```
#include <stdio.h>
int main()
{
    char c = 'a' ;
    int x = 12 ;
    double d = 3.141592 ;
    char s[30] = "hola" ;

    printf("%4d", x) ;          // SALIDA: ##12

    printf("%04d", x) ;         // SALIDA: 0012 (Decimal)
    printf("%04x", x) ;         // SALIDA: 000c (Hexadecimal)
    printf("%04o", x) ;         // SALIDA: 0014 (Octal)

    printf("%12.4g", d) ;       // SALIDA: #######3.142 (4 dígitos)

    printf("%12.4f", d) ;       // SALIDA: ######3.1416 (4 decimales)

    printf("%12.4e", d) ;       // SALIDA: ##3.1416e+00 (4 decimales)

    printf("%-6s", s) ;         // SALIDA: hola## (Just. Izquierda)

    printf("%6c", c) ;          // SALIDA: #####a (Just. Derecha)

    printf("%#04x", x) ;        // SALIDA: 0x0c (Hexadecimal + Base)
    printf("\n");

    printf("%*.*g", 12, 4, d) ; // SALIDA: #######3.142 (Alternativa)
    printf("%4.3d", x) ;        // SALIDA: #012         (Mínimo)
    printf("%6.2s", s) ;        // SALIDA: ####ho       (Máximo)
    printf("\n");
}
```

5.4. Salida formateada de datos en C++

En C++, la salida de datos formateada está orientada al modo de operación. A continuación se muestra un ejemplo de salida formateda de datos en C++, donde el símbolo **#** representa el espacio en blanco:

```
#include <iostream>
#include <iomanip>
int main()
{
    char c = 'a' ;
    int x = 12 ;
    double d = 3.141592 ;
    char s[30] = "hola" ;

    cout << setw(4) << x ;                      // SALIDA: ##12

    cout << setfill('0') ;                      // Modo Relleno Cero
    cout << dec << setw(4) << x ;               // SALIDA: 0012 (Decimal)
    cout << hex << setw(4) << x ;               // SALIDA: 000c (Hexadecimal)
    cout << oct << setw(4) << x ;               // SALIDA: 0014 (Octal)
    cout << setfill(' ') << dec ;               // Restaura Modo (Esp + Dec)

    cout << setprecision(4);                    // Establece Precisión 4
    cout << setw(12) << d ;                     // SALIDA: #######3.142 (4 dígitos)

    cout.setf(ios::fixed, ios::floatfield);     // Modo Fijo (F)
    cout << setw(12) << d ;                     // SALIDA: ######3.1416 (4 decimales)

    cout.setf(ios::scientific, ios::floatfield);// Modo Científico (E)
    cout << setw(12) << d ;                     // SALIDA: ##3.1416e+00 (4 decimales)

    cout.unsetf(ios::floatfield);               // Restaura Modo por Defecto (G)

    cout.setf(ios::left, ios::adjustfield);     // Modo Just. Izquierda
    cout << setw(6) << s ;                      // SALIDA: hola## (Just. Izquierda)
    cout.setf(ios::right, ios::adjustfield);    // Modo Just. Derecha
    cout << setw(6) << c ;                      // SALIDA: #####a (Just. Derecha)

    cout.setf(ios::internal, ios::adjustfield); // Modo Just. Interno
    cout.setf(ios::showbase);                   // Modo Mostrar-Base
    cout << setfill('0') << hex                 // Modo Relleno Cero + Hex
         << setw(4) << x                        // SALIDA: 0x0c (Hexadecimal + Base)
         << dec << setfill(' ') ;               // Restaura Modo (Esp + Dec)
    cout.unsetf(ios::showbase);                 // Restaura Modo por Defecto
    cout.unsetf(ios::adjustfield);              // Restaura Modo por Defecto

    cout.precision(4);  // Alternativa para establecer la precisión
    cout.fill('0');     // Alternativa para establecer el carácter de relleno

    cout << endl;
}
```

5.5. Comparativa entre ambos mecanismos

Consideramos que el uso de los flujos `cin` y `cout` para efectuar la Entrada/Salida básica en C++ es más simple y permite hacer programas más extensibles y robustos que mediante `printf` y `scanf` en C, por lo siguientes motivos:

- Es *más simple*, porque proporciona un mecanismo uniforme en el que no hay que considerar las peculiaridades asociadas al tipo de cada valor a leer o a escribir. Desde este punto de vista es más adecuado para el aprendizaje, ya que se puede comenzar a practicar sin tener que considerar pequeños detalles como, por ejemplo, el *especificador* concreto de la cadena de formato a utilizar en función del tipo de dato a leer y a escribir (nótese la diferencia de especificador entre `printf` y `scanf` para el tipo de datos `double`). Véase ejemplos en las secciones (5.1 y 5.2).

- Es *más simple*, porque el uso de `scanf` requiere que los parámetros usados sean punteros (direcciones de memoria) a las variables que almacenarán los valores leídos. Ello requiere el conocimiento de un concepto complejo en una fase inicial de aprendizaje donde aún no se está preparado para ello. Alternativamente, se podría utilizar la dirección de memoria de una variable sin conocer su semántica. En este caso se podría plantear el esquema indicando que para leer un dato hay que escribir `&` precediendo a la variable a leer con `scanf`. A priori esta segunda opción no resulta muy adecuada, y se ve dificultada por el hecho de que dicho operador `&` no se utiliza uniformemente para todos los tipos (se debe usar en variables de tipos simples y no usar en variables de tipo array de char). Además, es frecuente que en un contexto de aprendizaje de la programación, se olvide de poner dicho `&` (o al contrario, se incluya innecesariamente en el caso de variables de tipo array), en cuyo caso se genera un error que en las primeras sesiones prácticas aparece frecuentemente y complica el aprendizaje innecesariamente.

```
int main()
{
    int x ;
    char nombre[30] ;

    scanf("%d %s\n", &x, nombre) ;

    cin >> x >> nombre ;
}
```

- Otro problema importante con `scanf` es la necesidad de evitar el "desbordamiento de memoria" (*buffer overflow*) controlando el número de caracteres leídos. Ello obliga a que dicho número esté escrito en la cadena de formato, y no pueda depender del valor de una constante, que podría tomar otro valor. Por ejemplo:

```
#include <stdio.h>
#define MAX 20
int main()
{
    char nombre[MAX] ;
    scanf(" %19s", nombre) ;
    printf(" nombre: %s\n", nombre) ;
}
```

  El posterior cambio de la constante `MAX` a otro nuevo valor (por ejemplo 10) es una fuente de errores, ya que, en general, se considera que el cambio del valor de una constante no debe afectar a la corrección del programa. Como consecuencia ello obliga a revisar todo el código, y cambiar todos los números relacionados, en todas las partes del programa.

  En C++ este problema no existe en el caso del tipo `string`, y en el caso del array de char, quedaría así de forma adecuada:

```
#include <iostream>
#include <iomanip>
const int MAX = 20 ;
int main()
{
    char nombre[MAX] ;
    cin >> setw(MAX) >> nombre ;
    cout << "nombre: " << nombre << endl ;
}
```

- Favorece la creación de *programas más extensibles*, ya que no es necesario tener que mantener la correspondencia del tipo de una variable con la cadena de formato de una sentencia de E/S. Ello facilita la introducción de futuras modificaciones y disminuye la posibilidad de introducir nuevos errores. Por ejemplo, si queremos modificar el siguiente programa:

C:

```
typedef struct Dato {
    int   a ;
    float b ;
} Dato ;
int main()
{
    Dato d = { 1, 3.14 } ;
    printf("%d %g\n", d.a, d.b) ;
}
```

C++:

```
struct Dato {
    int   a ;
    float b ;
} ;
int main()
{
    Dato d = { 1, 3.14 } ;
    cout << d.a << ' ' <<  d.b << endl ;
}
```

para que el campo `a` pase a ser de fipo `float`, entonces en el caso de `printf` es necesario modificar la cadena de formato, mientras que usando `cout` no hay que hacer ningún cambio.

C:

```
typedef struct Dato {
    float a ;
    float b ;
} Dato ;
int main()
{
    Dato d = { 1, 3.14 } ;
    printf("%g %g\n", d.a, d.b) ;
}
```

C++:

```
struct Dato {
    float a ;
    float b ;
} ;
int main()
{
    Dato d = { 1, 3.14 } ;
    cout << d.a << ' ' <<  d.b << endl ;
}
```

- Es *más robusta*, porque elimina la posibilidad de que se omita, tanto por despiste como por desconocimiento, el operador `&` de las variables a leer con `scanf`, elimina posibles errores futuros por cambios en los tipos de las variables y además, evita determinados errores como el que se produciría en el siguiente programa dependiendo de que sea compilado en una máquina de 32 bits o de 64 bits[2]

```
#include <stdio.h>
int main()
{
    printf("sizeof(char): %u\n", sizeof(char)) ;
}
```

Nota: para solventar el problema indicado anteriormente (basado en el estándar ANSI C89), en el estándar C99 existe el *especificador* `z` para el tipo `size_t`:

```
#include <stdio.h>
int main()
{
    printf("sizeof(char): %z\n", sizeof(char)) ;
}
```

Aunque se trata de un ejemplo que puede parecer extremo, es una indicación de que la salida de datos homogénea con `cout` es más adecuada que con `printf`.

[2] Ver apéndice A

```
#include <iostream>
int main()
{
    cout << "sizeof(char): " << sizeof(char) << endl ;
}
```

6. Definición de Tipos

6.1. El Tipo Lógico (Booleano)

6.1.1. El Tipo Lógico (Booleano) en C (Estándar ANSI C89)

El lenguaje de programación C, según el estándar ANSI C89, no define ningún tipo predefinido para representar valores lógicos, y utiliza para ello el tipo `int`, utilizando un valor igual a cero (`0`) para representar el valor lógico *falso*, y cualquier valor distinto de cero para representar el valor lógico *verdadero*. Por ejemplo:

```
int main()
{
    int x = 7 ;
    int ok = 1 && (x > 5) || 0 ;
}
```

Esta notación está desaconsejada, especialmente en un entorno académico, en un contexto de aprendizaje de los conceptos fundamentales de programación, ya que no refleja adecuadamente los conceptos involucrados y puede interferir en el aprendizaje del uso de expresiones lógicas en las estructuras de control. Por ello, se recomienda seguir un enfoque donde se defina un nuevo tipo para representar los valores lógicos, así como la definición de constantes que reflejen adecuadamente dichos valores. por ejemplo:

```
typedef enum { FALSE, TRUE } bool_t ;
int main()
{
    int x = 7 ;
    bool_t ok = TRUE && (x > 5) || FALSE ;
}
```

Sin embargo, esta definición podría *colisionar* con otras definiciones realizadas dentro de bibliotecas que se estuviesen utilizando, por lo que hay que extremar las precauciones en este esquema.

6.1.2. El Tipo Lógico (Booleano) en C (Estándar C99)

El lenguaje de programación C, según el estándar C99, define el tipo predefinido `bool` para representar valores lógicos, y utiliza el fichero de la biblioteca estándar `<stdbool.h>` para definir el tipo `bool`, así como las constantes `false` para representar el valor lógico *falso*, y `true` para representar el valor lógico *verdadero*. Por ejemplo:

```
#include <stdbool.h>
int main()
{
    int x = 7 ;
    bool ok = true && (x > 5) || false ;
}
```

6.1.3. El Tipo Lógico (Booleano) en C++

El lenguaje de programación C++ dispone del tipo predefinido `bool` para representar los valores lógicos, así como las constantes predefinidas `true` y `false` para representar los valores lógicos *verdadero* y *falso*, de tal forma que la evaluación de expresiones lógicas produce como resultado un valor de dicho tipo.

```
int main()
{
    int x = 7 ;
    bool ok = true && (x > 5) || false ;
}
```

6.2. El Tipo Enumerado

6.2.1. El Tipo Enumerado en C

El lenguaje de programación C permite al programador definir tipos enumerados. El nombre del tipo enumerado se define dentro de un ámbito restringido de tipos enumerados, y por lo tanto requiere especificar la palabra reservada `enum` antes de utilizar el tipo. Además, existe una conversión implícita directa y automática entre valores del tipo enumerado y el tipo `int` y viceversa. Para definir el nombre del tipo enumerado dentro del ámbito global, es necesario utilizar la palabra reservada `typedef`. Por ejemplo:

Ámbito Restringido:

```
enum Color {
   Rojo, Verde, Azul
} ;
int main()
{
    enum Color c = Rojo ;
    c = c + 1 ;
}
```

Ámbito Global:

```
typedef enum {
   Rojo, Verde, Azul
} Color ;
int main()
{
    Color c = Rojo ;
    c = c + 1 ;
}
```

6.2.2. El Tipo Enumerado en C++

El lenguaje de programación C++ permite al programador definir tipos enumerados de una forma similar al lenguaje de programación C, con algunas diferencias:

- El nombre del tipo definido pertenece al ámbito donde haya sido definido el tipo enumerado, y puede ser utilizado directamente en dicho ámbito.
- Aunque tanto en C como en C++ se puede convertir de forma implícita y automática un valor de la enumeración al tipo `int`, ambos lenguajes se diferencian en que la conversión en la dirección contraria (es decir, de `int` al tipo enumerado) no ocurre de forma implícita (automática) en C++, requiriendo de un casting explícito. Esto hace que C++ sea más estricto y proporcione mayor seguridad en la gestión de los tipos.

Por ejemplo:

```
enum Color {
   Rojo, Verde, Azul
} ;
int main()
{
    Color c = Rojo ;
    c = Color(c + 1) ;
}
```

El nuevo estándar de C++ (C++11) define también otras clases de tipos enumerados con mayor control del ámbito de los valores definidos, pero no serán considerados en este documento.

6.3. Registros

6.3.1. Registros en C

El lenguaje de programación C permite al programador definir tipos estructurados (registros) como composición de elementos (campos), de tal forma que el nombre del tipo registro se define dentro de un ámbito restringido de tipos registro, y por lo tanto requiere especificar la palabra reservada `struct` antes de utilizar el tipo. Para definir el nombre del tipo registro dentro del ámbito global, es necesario utilizar la palabra reservada `typedef`. Por ejemplo:

Ámbito Restringido:

```
struct Fecha {
    int dia, mes, anyo ;
} ;
int main()
{
    struct Fecha hoy = { 25, 4, 2012 } ;
    printf("%d/%d/%d\n",
           hoy.dia, hoy.mes, hoy.anyo) ;
}
```

Ámbito Global:

```
typedef struct Fecha {
    int dia, mes, anyo ;
} Fecha ;
int main()
{
    Fecha hoy = { 25, 4, 2012 } ;
    printf("%d/%d/%d\n",
           hoy.dia, hoy.mes, hoy.anyo) ;
}
```

6.3.2. Registros en C++

El lenguaje de programación C++ permite definir al programador tipos estructurados (registros) de una forma similar al lenguaje de programación C, con la siguiente diferencia:

- El nombre del tipo definido pertenece al ámbito donde haya sido definido el tipo registro, y puede ser utilizado directamente en dicho ámbito.

Por ejemplo:

```
struct Fecha {
    int dia, mes, anyo ;
} ;
int main()
{
    Fecha hoy = { 25, 4, 2012 } ;
    cout << hoy.dia << '/' << hoy.mes << '/' << hoy.anyo << endl;
}
```

6.4. Arrays

En general, el tipo *array* se utiliza para definir una secuencia de un número determinado (definido en tiempo de compilación) de elementos de un mismo tipo de datos (simple o compuesto), de tal forma que se puede acceder a cada elemento individual de la secuencia de forma *parametrizada* mediante índices (el valor cero indexa el primer elemento de la colección).

Tanto en C como en C++ se pueden utilizar arrays predefinidos, pero en C++, desde la versión del estándar C++11 (previamente en *TR1*) es posible, además, usar el tipo `array` de la biblioteca estándar de C++.

6.4.1. Arrays predefinidos en C y C++

En ambos lenguajes se pueden usar arrays predefinidos de igual forma y con los mismos operadores. En este tipo de arrays:

- Sólo es posible inicializar una variable a partir de una enumeración de valores constantes, no es posible inicializarla a partir de otra variable.
- No es posible utilizar el operador de asignación para asignar directamente valores de tipo array a variables de tipo array.
- No es posible utilizar los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`).
- No se comprueba en *Tiempo de Ejecución* que los accesos a los elementos de un array sean correctos y se encuentren dentro de los límites válidos del array predefinido, por lo que será responsabilidad del programador comprobar que así sea.

C:

```
#define MAX    5
typedef int Vector[MAX] ;
int main()
{
    int i;
    Vector v = { 1, 2, 3, 4, 5 } ;
    for (i = 0; i < MAX; ++i) {
        v[i] = v[i] * 2 ;
    }
}
```

C++:

```
const int MAX = 5 ;
typedef int Vector[MAX] ;
int main()
{
    int i;
    Vector v = { 1, 2, 3, 4, 5 } ;
    for (i = 0; i < MAX; ++i) {
        v[i] = v[i] * 2 ;
    }
}
```

En las secciones 7.1.3 y 7.2.3 se describe el paso de parámetros de arrays predefinidos en C y C++ respectivamente.

6.4.2. El Tipo `array` de la biblioteca estándar de C++

El tipo `array` se incluyó en el estándar TR1 (Technical Report 1) de C++ (extensión del estándar de C++03) en 2005, y se incorporó al estándar de C++ en 2011 (C++11). Aunque los compiladores actuales lo tienen incorporado en la librería TR1, las nuevas versiones que soporten el nuevo estándar (C++11) lo traerán incorporado en las librerías de base. Si se utiliza un compilador que soporte el estándar C++11 se debe incluir la biblioteca estándar `<array>` y utilizar el espacio de nombres de `std`, mientras que si se utiliza un compilador que soporte el TR1 habrá que incluir la biblioteca `<tr1/array>` y utilizar el espacio de nombres de `std::tr1`.

C++11:

```
#include <array>
using namespace std ;
const int MAX = 5 ;
typedef array<int, MAX> Vector ;
int main()
{
    Vector v = { 1, 2, 3, 4, 5 } ;
    for (unsigned i = 0; i < v.size(); ++i) {
        v[i] = v[i] * 2 ;
    }
    Vector aux = v;
    aux = v;
    if (aux <= v) { /* ... */ }
}
```

C++03 + TR1:

```
#include <tr1/array>
using namespace std::tr1 ;
const int MAX = 5 ;
typedef array<int, MAX> Vector ;
int main()
{
    Vector v = {{ 1, 2, 3, 4, 5 }} ;
    for (unsigned i = 0; i < v.size(); ++i) {
        v[i] = v[i] * 2 ;
    }
    Vector aux = v;
    aux = v;
    if (aux <= v) { /* ... */ }
}
```

El tipo `array` proporciona operadores predefinidos para la copia y asignación (`=`), para las operaciones relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`)[3] y para acceder al número de elementos que componen un array (`size()`). Además, se permite la devolución de valores de tipo `array` desde funciones y es posible chequear si el acceso a elementos del array utiliza índices dentro del rango definido para el mismo.

Nótese que, en caso de utilizar un compilador que no soporte el nuevo estándar C++11, la inicialización de variables de tipo `array` de TR1 se deberá hacer con la enumeración de los valores constantes entre llaves dobles.

En la sección 7.2.4 se describe el paso de parámetros de arrays de la biblioteca estándar de C++.

6.4.3. Comparativa entre ambos mecanismos

La introducción del tipo `array` de la biblioteca estándar de C++ aporta numerosas ventajas respecto al uso de arrays predefinidos:

- Permite disponer de un tipo de datos que es consistente con el resto de tipos y dispone de los operadores predefinidos con los comportamientos habituales. Ello facilita la inicialización y copia de arrays mediante el operador de asignación, la devolución de arrays por funciones, y el uso de los operadores relacionales con el comportamiento esperado, como ocurre en el resto de tipos.
- Permite determinar el número de elementos del array con el operador `size()`, en lugar de hacerlo mediante la constante empleada en la definición del array. Ello disminuye el acoplamiento en el programa respecto a un valor constante global, y hace que el programa sea más robusto.
- Mejora las facilidades de depuración y localización de errores, ya que hace posible la detección de accesos fuera de rango a elementos del array mediante el flag de compilación `-D_GLIBCXX_DEBUG` [4].

Por el contrario, el tratamiento de los arrays predefinidos en C y C++ no es consistente con el tratamiento de los otros tipos de datos y presenta además algunas anomalías. Por ejemplo:

- No es posible la asignación directa de variables de tipo array predefinido, ni su devolución por parte de una función. Sin embargo, si el array predefinido se encuentra dentro de un registro, el registro si se puede asignar y devolver.
- La aplicación de los operadores relaciones produce resultados inesperados.
- El paso de parámetros de arrays predefinidos (sin utilizar el paso por referencia de C++) es propenso a errores no detectables en tiempo de compilación (no comprueba que el número de elementos de los parámetros sea correcto, así como permite la utilización del operador de asignación con resultados inesperados). Véase 7.1.3.

[3] En este caso, es necesario que los operadores relacionales estén definidos para el tipo base del array.

[4] Disponible en versiones de GCC posteriores a Noviembre de 2012 (versión 4.8). También se puede incorporar de forma fácil a las versiones actuales (instalando `http://www.lcc.uma.es/%7Evicente/docencia/cpplibs/array_tr1.zip`)

- Los arrays predefinidos tienen una conversión implícita automática al tipo puntero que es indeseable en muchos casos.

Finalmente, el tipo `array` de la biblioteca estándar de C++ proporciona tanto el constructor por defecto como el constructor de copia y el operador de asignación, por lo que podrá ser utilizado adecuadamente tanto en entornos de *clases* (`class`) como en entornos de programación genérica (`templates`), así también como tipo base de los contenedores estándares (`vector`, `deque`, `stack`, `list`, etc.). Esto no es posible con los arrays predefinidos.

Clases:

```
class Pixel {
public:
    Pixel() : rgb() {}
    Pixel(const Pixel& o)
        : rgb(o.rgb) {}
    Pixel& operator=(const Pixel& o)
    {
        rgb = o.rgb ;
        return *this ;
    }
private:
    array<short, 3> rgb ;
    // short rgb[3] ;     // ERROR
} ;
```

Genericidad:

```
template <typename Tipo>
void intercambio(Tipo& x, Tipo& y)
{
    Tipo aux = x ;
    x = y ;
    y = aux ;
}

typedef array<short, 3> RGB ;
// typedef short RGB[3] ;     // ERROR
int main()
{
    RGB rgb1, rgb2 ;
    vector<RGB> imagen ;
    Dato<RGB> dat ;
    intercambio(rgb1, rgb2) ;
    imagen.push_back(rgb1) ;
    dat.set(rgb1) ;
    rgb2 = dat.get() ;
}
```

Clases Genéricas:

```
template <typename TIPO>
class Dato {
public:
    Dato() : dat() {}
    Dato(const Dato& o)
        : dat(o.dat) {}
    Dato(const TIPO& o)
        : dat(o) {}
    Dato& operator=(const Dato& o)
    {
        dat = o.dat ;
        return *this ;
    }
    void set(const TIPO& o)
    {
        dat = o ;
    }
    TIPO get() const
    {
        return dat ;
    }
private:
    TIPO dat ;
} ;
```

En contra del uso del tipo `array` se puede argumentar que, aunque forma parte del estándar de C++, es posible que algunos compiladores aún no lo incorporen. Sin embargo, los compiladores más usados si lo incorporan (incorporado en 2005 a la versión 4.0 de GCC en TR1, y en 2007 a la versión 4.3 de GCC como soporte experimental para C++11 en la biblioteca estándar) y es, además, fácilmente incorporable a otros compiladores más antiguos.

6.5. Cadenas de Caracteres

6.5.1. Cadenas de caracteres en C

En C las cadenas de caracteres se gestionan mediante arrays predefinidos de caracteres (tipo `char`) de un determinado tamaño especificado en tiempo de compilación. Una cadena está formada por la secuencia de caracteres comprendidos entre el principio del array y un carácter terminador especial, que marca el

final de la misma (el carácter `'\0'`). Por lo tanto, siempre habrá que considerar que el array tenga espacio suficiente para almacenar dicho carácter terminador. Las cadenas de caracteres constantes literales se expresan como una secuencia de caracteres entre comillas dobles (`"`).

Al estar definidas como arrays predefinidos de caracteres, todo lo explicado anteriormente respecto a los arrays predefinidos (véase 6.4.1) también se aplica a las cadenas de caracteres. En la sección 7.1.4 se describe el paso de parámetros de cadenas de caracteres en C.

6.5.2. El Tipo `string` de la biblioteca estándar de C++

El tipo `string` de la biblioteca estándar de C++ permite representar cadenas de caracteres de *longitud finita*, limitada por la implementación. Para ello, se debe incluir la biblioteca estándar `<string>`, así como utilizar el espacio de nombres de `std`.

Es posible asignar (`=`) un *string* a variables de dicho tipo, usar los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`) con el comportamiento esperado y devolver un *string* desde una función. El paso de parámetros, tanto por valor como por referencia y referencia constante, es robusto y mantiene su semántica previamente establecida.

Además, es posible acceder al número de elementos que lo componen mediante el operador `size()`, acceder a los caracteres que lo componen mediante el operador `[i]`, y utilizar el flag de compilación `-D_GLIBCXX_DEBUG` (en GCC) para generar código en el que se compruebe que en los accesos a posiciones de un string se utiliza un índice que está dentro del rango correcto.

También es posible utilizar operadores (`+`, `+=`) para concatenar cadenas y caracteres y otras operaciones útiles, como la obtención de una *nueva* subcadena (de tipo `string`) a partir de un índice `i`, con un tamaño especificado por `sz`. Por ejemplo:

```
#include <string>
using namespace std;
int main()
{
    string nombre = "pepe" ;
    for (unsigned i = 0; i < nombre.size(); ++i) {
        nombre[i] = nombre[i] - 'a' + 'A' ;
    }
    nombre += "Luis" ;
    string aux = nombre.substr(5, 2) ;
    if (aux <= nombre) { /* ... */ }
}
```

En la sección 7.2.5 se describe el paso de parámetros del tipo `string` de la biblioteca estándar de C++.

6.5.3. Comparativa entre ambos mecanismos

La definición de cadenas de caracteres del tipo `string` permite trabajar con cadenas de caracteres más robustas y con mejores características que las cadenas de caracteres predefinidas de C. Algunas ventajas del uso del tipo `string` de la biblioteca estándar de C++ respecto al uso de `array` de `char` son:

- Es más simple, ya que dispone de operadores predefinidos (`=`, `==`, `!=`, `>`, `>=`, `<`, `<=`) cuyo comportamiento es el esperado y no requiere considerar casos

especiales. En el caso de `array` de `char` los operadores relacionales no proporcionan los resultados esperados y la asignación sólo está permitida en la inicialización y si el elemento a asignar es un literal constante o es un campo de un registro que se asigna a una variable de tipo registro.

- Es consistente con los otros tipos de datos, permitiendo la asignación (`=`) de variables de dicho tipo, el uso de los operadores relacionales con el comportamiento esperado y su devolución desde una función.
- No requiere definir a priori un tamaño en *tiempo de compilación* para cada posible cadena de caracteres que utilice nuestro programa.
- No es necesario reservar espacio para el carácter terminador ni considerarlo en cualquier manipulación.
- Permite acceder al número de elementos que lo componen mediante el operador `size()`.
- Ofrece una entrada/salida más limpia y segura, sin posibilidad de *"desbordamiento de memoria"* (*buffer overflow*).
- Ofrece operadores predefinidos robustos para concatenación o para añadir caracteres. Los correspondientes de `array` de `char` son propensos a *"desbordamientos de memoria"* (*buffer overflow*).
- El paso de parámetros del tipo `string` es consistente con la semántica de paso por valor, referencia y referencia constante, sin embargo, en el caso de array de `char`, se debe realizar como array abierto (constante o no) o puntero a `char`, con las inconsistencias que ello comporta, por ejemplo permite el operador de asignación (`=`) y relacionales con resultados inesperados.
- No se puede declarar un array de caracteres dentro de una función, y que la función retorne ese array. Esto provoca errores en tiempo de ejecución complicados. Por ejemplo:

```
#include <stdio.h>
#include <string.h>
typedef char Cadena[100] ;
char* proc_1()
{
    Cadena res = "hola" ;
    return res ;
}
void proc_2(char* c)
{
    c = proc_1() ;
}
void proc_3(char* c)
{
    strcpy(c, proc_1()) ;
}
int main()
{
    Cadena c ;
    proc_2(c) ;
    printf("%s\n", c) ;
    proc_3(c) ;
    printf("%s\n", c) ;
}
```

7. Subprogramas: Paso de Parámetros

7.1. Subprogramas y paso de parámetros en C

7.1.1. Paso de parámetros de tipos simples en C

El lenguaje de programación C sólo define el *paso por valor* de parámetros, por lo que para soportar parámetros de salida (o entrada/salida) el programador debe utilizar explícitamente otros mecanismos, tales como el uso del tipo *puntero*, el operador de *dirección de memoria* (`&`) y los operadores de *desreferenciación* (`*` y `->`).

```
int mayor(int x, int y)
{
    return (x > y) ? x : y ;
}
void intercambiar(int* x, int* y)
{
    int aux = *x ;
    *x = *y ;
    *y = aux ;
}
void proc(int* z)
{
    int i = mayor(5, *z) ;
    intercambiar(&i, z) ;
}
int main()
{
    int x = 5 ;
    int y = mayor(9, x) ;
    intercambiar(&x, &y) ;
    proc(&x) ;
}
```

Nótese cómo en el ejemplo se muestra que en determinadas ocasiones se debe utilizar el operador de *dirección* (`&`) y el operador de *desreferenciación*, y en determinadas ocasiones no se deben utilizar (invocaciones a `mayor` e `intercambiar` desde dentro de `proc` y `main`).

La complejidad de este mecanismo hace que no sea adecuado en un contexto de aprendizaje de la programación, ya que requiere el dominio de conceptos complejos que deben ser introducidos en una etapa temprana del aprendizaje, dificultando el aprendizaje de los conceptos de programación modular y uso de subprogramas.

7.1.2. Paso de parámetros de tipo registro en C

El uso de registros puede llevar asociado el uso de estructuras de datos que requieran un considerable uso de memoria. En este caso, su paso por valor puede suponer una gran sobrecarga (tanto en tiempo como en memoria).

```
struct Imagen {
    int nfils, ncols ;
    int img[1000][1000] ;
} ;
int media(struct Imagen img1, int f, int c)
{
    /* en este punto, 4 imagenes de (1000x1000) se     */
    /* encuentran almacenadas en la Pila de ejecucion  */
    /* con la posibilidad de desborde de la pila       */
}
void transformar(struct Imagen img1, struct Imagen* img2)
{
    int f, c ;
```

```
    img2->nfils = img1.nfils ;
    img2->ncols = img1.ncols ;
    for (f = 0 ; f < img2->nfils ; ++f) {
        for (c = 0 ; c < img2->ncols ; ++c) {
            img2->img[f][c] = media(img1, f, c) ;
        }
    }
}
int main()
{
    struct Imagen img1, img2 ;
    /* ... */
    transformar(img1, &img2) ;
}
```

Por este motivo, en C los tipos estructurados siempre se pasarán como *punteros*, que en el caso de parámetros de entrada deberán ser *punteros a constantes*. Por ejemplo:

```
struct Imagen {
    int nfils, ncols ;
    int img[1000][1000] ;
} ;
int media(const struct Imagen* img1, int f, int c)
{
    /* ... */
}
void transformar(const struct Imagen* img1, struct Imagen* img2)
{
    int f, c ;
    img2->nfils = img1->nfils ;
    img2->ncols = img1->ncols ;
    for (f = 0 ; f < img2->nfils ; ++f) {
        for (c = 0 ; c < img2->ncols ; ++c) {
            img2->img[f][c] = media(img1, f, c) ;
        }
    }
}
int main()
{
    struct Imagen img1, img2 ;
    /* ... */
    transformar(&img1, &img2) ;
}
```

7.1.3. Paso de parámetros de arrays predefinidos en C

En C no es posible pasar arrays *por valor*. Al invocar a un subprograma con un parámetro de tipo array se pasa automáticamente la *dirección* de dicho array. Posteriormente, cuando se intenta acceder en el cuerpo del subprograma a los elementos del array, se realiza (de forma automática y transparente) una *desreferencia* que permite acceder a la posición de memoria adecuada.

```
#define MAX 10
typedef int Datos[MAX] ;
void inicializar(Datos d)
{
    int i ;
    for (i = 0 ; i < MAX ; ++i) {
        d[i] = i ;
    }
}
int suma(Datos d)
{
    int i, s = 0 ;
    for (i = 0 ; i < MAX ; ++i) {
        s += d[i] ;
    }
    return s ;
```

```
}
int main()
{
    Datos d ;
    inicializar(d) ;
    printf("%d\n", suma(d) ) ;
}
```

Nótese cómo este paso de parámetros de arrays (aparentemente un *paso por valor*) actúa de forma implícita como un *paso por referencia*, permitiendo que desde dentro del subprograma se modifique el parámetro. Para evitar esta circunstancia, cuando se desee pasar un array como parámetro de entrada a un subprograma se debe cualificar el parámetro con `const`, consiguiendo así que el compilador nos avise de accesos incorrectos al parámetro en los que se intente modificar alguna posición del array. Por ejemplo:

```
int suma(const Datos d)
{
    int i, s = 0 ;
    for (i = 0 ; i < MAX ; ++i) {
        s += d[i] ;
    }
    return s ;
}
```

También es posible realizar el paso de parámetros de *arrays abiertos* (en caso de múltiples dimensiones, sólo se permite dejar abierta la primera dimensión), por ejemplo:

```
int copiar(int d[], int nd, const int o[], int no)
{
    int i, m = menor(nd, no) ;
    for (i = 0 ; i < m ; ++i) {
        d[i] = o[i] ;
    }
    return m ;
}
```

El paso de parámetros de arrays en C está sujeto a numerosas peculiaridades que hacen de su tratamiento un caso excepcional con respecto a los otros tipos y es propenso a introducir numerosos errores de programación (especialmente en un contexto de aprendizaje a la programación), por lo que exige un tratamiento cauteloso por parte del programador.

- No se comprueba que los parámetros actuales y formales sean del mismo tipo (tamaño), *siendo posible* pasar como parámetro actual un array de un tamaño diferente al especificado como parámetro formal, con los errores que ello conlleva. Por ejemplo:

  ```
  #define NELMS_1    9
  #define NELMS_2    5
  typedef int Vector_1[NELMS_1] ;
  typedef int Vector_2[NELMS_2] ;
  void copiar(Vector_1 destino, const Vector_1 origen)
  {
      int i ;
      for (i = 0 ; i < NELMS_1 ; ++i) {
          destino[i] = origen[i] ;
      }
  }
  int main()
  {
      Vector_1 v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
      Vector_2 v2 ;
      copiar(v2, v1) ;  /* Error al pasar una variable de tipo Vector_2 */
  }
  ```

- Si se utiliza el operador de asignación (=), no se produce ningún error de compilación. Este comportamiento es diferente si se utiliza la asignación entre arrays que no son parámetros. En el primer caso el compilador no nos avisa de ningún error (aunque en realidad obtendremos resultados inesperados) mientras que en el segundo si tendremos aviso de error. Por ejemplo:

```
#define MAX    5
typedef int Datos[MAX] ;
void copiar(Datos d, Datos o)
{
    d = o ;      /* No error de compilacion. resultados inesperados */
}
int main()
{
    Datos d2, d1 = { 1, 2, 3, 4, 5 } ;
    d2 = d1 ;    /* error: invalid array assignment */
    copiar(d2, d1) ;
}
```

- Comportamiento inesperado cuando los cambios realizados en un parámetro, aparentemente pasado *por valor*, modifican el parámetro actual en la llamada.

```
#define MAX 10
typedef int Datos[MAX] ;
void inicializar(Datos d)
{
    int i ;
    for (i = 0 ; i < MAX ; ++i) {
        d[i] = i ;
    }
}
int main()
{
    Datos d ;
    inicializar(d) ;
}
```

7.1.4. Paso de parámetros de cadenas de caracteres en C

Las cadenas de caracteres en C se definen como arrays predefinidos de tipo base `char`, por lo que todo lo mencionado en la sección 7.1.3 anterior también es aplicable a cadenas de caracteres.

Cuando se trabaja con cadenas de carácteres es frecuente tratar con cadenas de carácteres almacenadas en arrays de diferentes tamaños. El procesamiento en subprogramas de dichas cadenas suele ser equivalente, por lo que frecuentemente es adecuado tratar los parámetros de tipo cadena de caráteres como arrays abiertos de tipo `char`. Ello permite disponer de subprogramas para manipular cadenas almacenadas en arrays de tamaños diferentes.

En este tipo de situaciones es importante que el subprograma reciba también el tamaño máximo que la cadena puede alcanzar. De esta forma podemos evitar acceder más allá de sus límites. Si la cadena se utiliza como parámetro de entrada no es necesario pasar el tamaño del array, ya que el final de la cadena viene marcado por el carácter terminador.

```
#include <stdio.h>
#define MAX_CADENA   10
typedef char Cadena[MAX_CADENA] ;
void escribir(const char cad[])
{
    int i ;
```

```
    for (i = 0; cad[i] != '\0'; ++i) {
        printf("%c",cad[i]) ;
    }
}
int longitud(const char cad[])
{
    int i = 0 ;
    while (cad[i] != '\0') {
        ++i ;
    }
    return i ;
}
void copiar(char destino[], const char origen[], int sz)
{
    int i ;
    for (i = 0; (i < sz-1)&&(origen[i] != '\0'); ++i) {
        destino[i] = origen[i] ;
    }
    destino[i] = '\0' ;
}
int main()
{
    Cadena c1 = "Pepe" ;
    Cadena c2 ;
    int l = longitud(c1) ;
    copiar(c2, c1, MAX_CADENA) ;
    escribir(c2) ;
    copiar(c2, "Luis", MAX_CADENA) ;
    escribir(c2) ;
}
```

Nota: también es posible realizar el paso de parámetros de cadenas de caracteres como `char*` y `const char*` con la misma semántica que la especificada anteriormente.

7.2. Subprogramas y paso de parámetros en C++

7.2.1. Paso de parámetros de tipos simples en C++

El lenguaje de programación C++ define el *paso por valor* y el *paso por referencia* de parámetros, de tal forma que los parámetros de entrada de *tipos simples* se pasarán *por valor*, y los parámetros de salida (o entrada/salida) se pasarán *por referencia*. Por ejemplo:

```
int mayor(int x, int y)
{
    return (x > y) ? x : y ;
}
void intercambiar(int& x, int& y)
{
    int aux = x ;
    x = y ;
    y = aux ;
}
void proc(int& z)
{
    int i = mayor(5, z) ;
    intercambiar(i, z) ;
}
int main()
{
    int x = 5 ;
    int y = mayor(9, x) ;
    intercambiar(x, y) ;
    proc(x) ;
}
```

El ejemplo muestra cómo toda la complejidad asociada del paso de parámetros *por referencia* es manejada de forma transparente por el compilador, primando

su facilidad y consistencia de uso (compárese con la complejidad del código equivalente en C 7.1.1).

7.2.2. Paso de Parámetros de tipo registro en C++

En C++ es posible pasar valores de tipo registro como parámetros *por valor*. Sin embargo, como se vio en una sección 7.1.2, ello conlleva una sobrecarga asociada (mayor consumo de memoria, y tiempo de ejecución en copiar los elementos), que puede ser considerable en el caso de estructuras grandes.

Por este motivo, en C++ los tipos estructurados siempre se pasarán *por referencia*, que en el caso de parámetros de entrada deberán ser *referencias constantes*. Por ejemplo:

```
struct Imagen {
    int nfils, ncols ;
    int img[1000][1000] ;
} ;
int media(const Imagen& img1, int f, int c)
{
    /* ... */
}
void transformar(const Imagen& img1, Imagen& img2)
{
    img2.nfils = img1.nfils ;
    img2.ncols = img1.ncols ;
    for (int f = 0 ; f < img2.nfils ; ++f) {
        for (int c = 0 ; c < img2.ncols ; ++c) {
            img2.img[f][c] = media(img1, f, c) ;
        }
    }
}
int main()
{
    Imagen img1, img2 ;
    /* ... */
    transformar(img1, img2) ;
}
```

7.2.3. Paso de parámetros de arrays predefinidos en C++

En C++ no es posible pasar valores que sean arrays predefinidos como parámetros *por valor*. Al igual que ocurre en C (ver sección 7.1.3) el paso de un array como parámetro supone el paso como parámetro de su dirección de memoria. A diferencia del mecanismo planteado en C para el paso de este tipo de parámetros, en C++ siempre pasaremos los arrays por *referencia*, empleando paso por *referencia constante* cuando el parámetro sea de *entrada*[5]. De esta forma conseguiremos un comportamiento más robusto que en el enfoque usado en C.

```
const int MAX = 10 ;
typedef int Datos[MAX] ;
void inicializar(Datos& d)
{
    for (int i = 0 ; i < MAX ; ++i) {
        d[i] = i ;
    }
}
int suma(const Datos& d)
{
    int s = 0 ;
    for (int i = 0 ; i < MAX ; ++i) {
        s += d[i] ;
```

[5]Nótese que este enfoque no podrá ser empleado en caso de *arrays abiertos*.

```
    }
    return s ;
}
int main()
{
    Datos d ;
    inicializar(d) ;
    printf("%d\n", suma(d) ) ;
}
```

En la sección 7.1.3 vimos cómo el paso de parámetros de tipo array en C tiene una serie de inconvenientes. El uso del paso por *referencia* para parámetros de tipo array proporciona una serie de ventajas que permiten superar dichos inconvenientes:

- Chequeo de tipos. Al usar el paso por referencia el compilador comprueba que los parámetros actuales y formales sean del mismo tipo, comprobando que ambos posean el mismo número de elementos, generando un error en caso contrario. Por ejemplo:

```
const int NELMS_1 = 9 ;
const int NELMS_2 = 5 ;
typedef int Vector_1[NELMS_1] ;
typedef int Vector_2[NELMS_2] ;
void copiar(Vector_1& destino, const Vector_1& origen)
{
    for (int i = 0 ; i < NELMS_1 ; ++i) {
        destino[i] = origen[i] ;
    }
}
int main()
{
    Vector_1 v1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
    Vector_2 v2 ;
    copiar(v2, v1) ;  /* Error de Compilación */
}
```

- Detección de errores por asignación de arrays. El compilador avisará mediante un mensaje de error en caso de que se intente utilizar el operador de asignación predefinido (=) para asignar un array a un parámetro de tipo array.

```
const int MAX = 5 ;
typedef int Datos[MAX] ;
void copiar(Datos& d, Datos& o)
{
    d = o ;      /* Error de Compilación */
}
int main()
{
    Datos d2, d1 = { 1, 2, 3, 4, 5 } ;
    d2 = d1 ;    /* error: invalid array assignment */
    copiar(d2, d1) ;
}
```

- Detección de intento de modificación de parámetros de entrada. El parámetro actual que se corresponda con un parámetro de entrada no debe verse afectado por ninguna modificación en el subprograma llamado. Utilizando paso por referencia constante conseguimos que el compilador avise de cualquier intento de alterar el parámetro formal correspondiente, obteniendo así un código más robusto.

```
const int MAX = 5 ;
typedef int Datos[MAX] ;
```

```
void inicializar(const Datos& d)
{
    for (int i = 0 ; i < MAX ; ++i) {
        d[i] = i ;   /* Error de Compilación */
    }
}
int main()
{
    Datos d ;
    inicializar(d) ;
}
```

7.2.4. Paso de parámetros del tipo `array` de la biblioteca estándar de C++

El tipo array de la biblioteca estándar es consistente con el resto de tipos de datos en otros aspectos y también lo es en el paso de parámetros. Por tanto, cuando se pasa por valor uno de estos arrays se realiza una copia del valor situado como parámetro actual. Por este motivo y para evitar la sobrecarga asociada al proceso de copia del valor pasado, cuando este tipo de arrays vayan a ser utilizados como valores de *entrada* a subprogramas deben pasarse por *referencia constante*. De igual forma, cuando vayan a ser utilizados como valores de *salida* o *entrada/salida* deben ser pasados por *referencia*. Por ejemplo:

```
#include <tr1/array>
using namespace std::tr1 ;
typedef array<int, 50> Datos ;
void copiar(Datos& destino, const Datos& origen)
{
    destino = origen ;
}
void escribir(const Datos& d)
{
    for (unsigned i = 0 ; i < d.size() ; ++i) {
        cout << d[i] << ' ' ;
    }
    cout << endl ;
}
int main()
{
    Datos datos_1, datos_2 = {{ 1, 2, 3, 4, 5 }} ;
    copiar(datos_1, datos_2) ;
    escribir(datos_1) ;
}
```

7.2.5. Paso de parámetros del tipo `string` de la biblioteca estándar de C++

Al igual que ocurre con los arrays de la biblioteca estándar y por los mismos motivos, cuando se deseen pasar valores de tipo `string` como estrada a subprogramas deberá hacerse por *referencia constante*. Asímismo, cuando se deseen pasar valores de tipo `string` como parámetros de salida o entrada/salida, deberá hacerse por *referencia*. Por ejemplo:

```
#include <string>
using namespace std ;
void copiar(string& destino, const string& origen)
{
    destino = origen ;
}
void escribir(const string& d)
{
    for (unsigned i = 0 ; i < d.size() ; ++i) {
        cout << d[i] << ' ' ;
```

```
    }
    cout << endl ;
}
int main()
{
    string nombre_1, nombre_2 = "hola" ;
    copiar(nombre_1, nombre_2) ;
    escribir(nombre_1) ;
}
```

8. El Tipo Puntero y la Gestión de Memoria Dinámica

8.1. El tipo puntero y la gestión de memoria dinámica en C

8.1.1. El tipo puntero en C

La lenguaje de programación C permite definir punteros a variables de tipos especificados. Esta definición se puede realizar tanto mediante la definición de un tipo simbólico:

```
typedef struct Nodo* PNodo ;
struct Nodo {
    PNodo sig ;
    int   dato ;
} ;
int main()
{
    PNodo lista = NULL ;
    /* ... */
}
```

como mediante la definición explícita de punteros:

```
struct Nodo {
    struct Nodo* sig ;
    int          dato ;
} ;
int main()
{
    struct Nodo* lista = NULL ;
    /* ... */
}
```

Dada las características de bajo nivel del lenguaje de programación C, en las cuales la utilización de punteros surge en muy diversas situaciones, la definición explícita de punteros suele ser más habitual, debido a su flexibilidad y capacidad de adaptarse más fácilmente a cada contexto de utilización. Nótese que, en C, la semántica del paso de parámetros por referencia recae en la utilización de punteros, por lo que cada tipo de datos es susceptible de ser utilizado como base para ser apuntado por un puntero.

```
struct Imagen {
    int nfils, ncols ;
    int img[1000][1000] ;
} ;
void transformar(const struct Imagen* img1, struct Imagen* img2)
{
    /* ... */
}
int main()
{
    struct Imagen img1, img2 ;
```

```
    /* ... */
    transformar(&img1, &img2) ;
}
```

Además de las operaciones de asignación (`=`) y comparación (`==` y `!=`) entre punteros del mismo tipo, C posee dos operadores de desreferenciación, el operador prefijo *asterisco* (`*`) permite desreferenciar un puntero no NULO, y acceder a todo el objeto apuntado, mientras que el operador postfijo *flecha* (`->`) permite desreferenciar un puntero no NULO, y acceder a un miembro del registro apuntado. Así mismo, el operador prefijo *ampersand* (`&`) permite obtener la dirección en memoria (valor de un puntero) de un determinado objeto.

```
struct Imagen {
    int nfils, ncols ;
    int img[1000][1000] ;
} ;
void transformar(const struct Imagen* img1, struct Imagen* img2)
{
    *img2 = *img1 ;
    /* ... */
    img2->nfils = 2 * img1->nfils ;
}
int main()
{
    struct Imagen img1, img2 ;
    /* ... */
    transformar(&img1, &img2) ;
}
```

8.1.2. La gestión de memoria dinámica en C

El lenguaje de programación C proporciona mecanismos para la gestión de la memoria dinámica mediante las siguientes funciones proporcionadas por la biblioteca estándar (`stdlib.h`).

- `void *malloc(size_t size)`: reserva un área de `size` bytes en la zona de memoria dinámica (*heap*), y devuelve un puntero a dicha área reservada.
- `void *calloc(size_t nelms, size_t size)`: reserva (e inicializa a cero) un área de memoria consecutiva en la zona de memoria dinámica (*heap*) para almacenar un **array** de `nelms` elementos de tamaño `size` bytes cada uno, y devuelve un puntero a dicha área reservada.
- `void *realloc(void *ptr, size_t size)`: cambia el tamaño del área de memoria dinámica apuntado por `ptr` para que pase a ocupar un nuevo tamaño de `size` bytes, y devuelve un puntero a la nueva área reservada. Si el área de memoria original puede ser expandida o contraída al nuevo tamaño `size`, entonces se expande o contrae al nuevo tamaño, sin necesidad de copiar los datos. Sin embargo, si el área de memoria original no puede ser expandida, entonces se reserva un nuevo área de memoria, se copian los datos originales almacenados en el área original, y se libera el área original. Finalmente se devuelve un puntero al área reservada.
- `void free(void *ptr)`: libera el área de memoria dinámica apuntada por `ptr`, que debe haber sido reservada previamente mediante una invocación a `malloc`, `calloc` o `realloc`.

El operador `sizeof(Tipo)` devuelve el tamaño en bytes necesario para almacenar una variable del tipo especificado. Si este operador se aplica a una determinada expresión, entonces devuelve el tamaño en bytes necesario para almacenar el resultado de dicha expresión.

```
struct Dato {
    int valor ;
} ;
int main()
{
    struct Dato* ptr = malloc(sizeof(*ptr)) ;
    if (ptr != NULL) {
        ptr->valor = 5 ;
        /* ... */
        free(ptr) ;
    }
}
```

8.1.3. El operador de dirección en C

El lenguaje de programación C permite que los punteros no sólo apunten a objetos en memoria dinámica, sino que también apunten a objetos alojados en cualquier parte de la memoria. Para ello, el operador prefijo *ampersand* (`&`) permite obtener la dirección en memoria donde se encuentra alojado un determinado objeto. Esta dirección en memoria se representa mediante un puntero a la zona de memoria donde se encuentra alojado el objeto. Este mecanismo se utiliza extensivamente en el paso de parámetros (véase 7.1), así como en muy diversas situaciones.

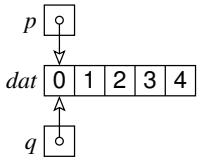
```
struct Dato {
    int valor ;
} ;

void incremento(int* val, int x)
{
    *val += x ;
}
int main()
{
    struct Dato dat ;
    struct Dato* ptr = &dat ;
    int* pv = &ptr->valor ;
    ptr->valor = 5 ;
    *pv += 3 ;
    incremento(&dat.valor, 7) ;  /* dat.valor toma el valor 15 */
}
```

Es importante hacer notar que en un ámbito de aprendizaje a la programación, es posible que la comprensión de los conceptos relacionados con los punteros y la gestión de memoria dinámica se vea dificultada si éstos son mezclados con los conceptos de punteros a objetos en otras zonas de memoria.

8.1.4. Arrays y aritmética de punteros en C

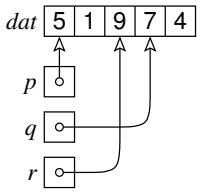
Cuando los arrays se pasan como parámetros a un subprograma, se produce una conversión automática, y lo que realmente se pasa es un puntero al primer elemento del array. Del mismo modo, cuando se asigna un array a un puntero, se produce una conversión automática y se asigna un puntero al primer elemento del array. Dicho de otra forma, siempre que se **utiliza** un array, se realiza una conversión automática y se genera un puntero con la dirección en memoria del primer elemento del array. Es importante resaltar que esta conversión automática de array a puntero **no significa** que arrays y punteros sean lo mismo, son conceptos y estructuras diferentes.

```
#define NELMS  5
typedef int Datos[NELMS] ;
int main()
{
    Datos dat = { 0, 1, 2, 3, 4 } ;
    int* p = dat ;
    int* q = &dat[0] ;
}
```



En C es posible utilizar aritmética de punteros, tanto para sumar (+) como para restar (-) un número entero a un puntero. El resultado es un puntero a un objeto del mismo tipo, pero situado tantas posiciones a la derecha o izquierda como indique el valor del número entero sumado o restado.

```
#define NELMS  5
typedef int Datos[NELMS] ;
int main()
{
    Datos dat = { 0, 1, 2, 3, 4 } ;
    int* p = dat ;
    int* q = p + 3 ;
    int* r = q - 1 ;
    *p = 5 ;
    *q = 7 ;
    *r = 9 ;
}
```



La siguiente table muestra algunas equivalencias entre operadores al tratar con arimética de punteros:

`++p`	⇔	`p = p + 1`
`--p`	⇔	`p = p - 1`
`p += n`	⇔	`p = p + n`
`p -= n`	⇔	`p = p - n`
`p[i]`	⇔	`*(p + i)`

8.1.5. Punteros genéricos en C

Además de la definición de punteros a tipos específicos, el lenguaje de programación C permite definir punteros a tipos desconocidos (genericidad de bajo nivel, sin control estricto de tipos), que no pueden ser desreferenciados. Para poder ser desreferenciados antes deben ser convertidos (casting) a punteros a un determinado tipo concreto. Las conversiones entre `void*` y otros tipos punteros son automáticas en C, y no es necesaria la conversión explícita.

```
struct Imagen {
    int nfils, ncols ;
    int img[1000][1000] ;
} ;
void cero(void* dato, unsigned sz)
{
    int i ;
    char* dat = (char*)dato ;
    for (i = 0; i < sz; ++i) {
        dat[i] = 0 ;
    }
}
int main()
{
    struct Imagen img ;
    cero(&img, sizeof(img)) ;
}
```

8.1.6. Tipos opacos en C

Así mismo, el lenguaje de programación C también permite la definición de tipos *opacos* mediante la definición de punteros a tipos incompletos (deben

ser registros), los cuales podrán ser definidos en módulos de implementación independientes. Estos tipos incompletos deberán ser definidos en el módulo de implementación antes de poder ser desreferenciados.

```
//- dato.h ----------------------------
#ifndef dato_h_
#define dato_h_
typedef struct DatoImpl* Dato ;
Dato crear() ;
void incrementar(Dato d, int x) ;
void destruir(Dato d) ;
#endif
//- dato.c ----------------------------
#include "dato.h"
struct DatoImpl {
    int valor ;
};
Dato crear()
{
    Dato ptr = malloc(sizeof(*ptr)) ;
    if (ptr != NULL) {
        ptr->valor = 0 ;
    }
    return ptr;
}
void incrementar(Dato d, int x)
{
    if (d != NULL) {
        d->valor += x ;
    }
}
void destruir(Dato d)
{
    free(d) ;
}
//- main.c ----------------------------
#include "dato.h"
int main()
{
    Dato dat ;
    dat = crear() ;
    incrementar(dat, 5) ;
    destruir(dat) ;
}
//---------------------------------------
```

8.2. El tipo puntero y la gestión de memoria dinámica en C++

8.2.1. El tipo puntero en C++

Aunque C++ soporta la semántica de punteros de forma similar a C (véase la sección 8.1.1), C++ es un lenguaje más fuertemente tipado que C y está orientado a la definición de abstracciones de más alto nivel. Además, C++ soporta directamente la semántica del paso de parámetros por referencia, por lo que no es necesario utilizar punteros para el paso de parámetros. Por todo ello, la utilización de punteros en C++ se restringe más a entornos de bajo nivel (encapsulados en abstracciones de mayor nivel) relacionados con la gestión de memoria dinámica, así como en entornos polimórficos orientados a objetos.

En estas circunstancias de punteros orientados a la gestión de memoria dinámica, dentro de un entorno académico de aprendizaje de la programación, suele ser más adecuado realizar una definición simbólica del tipo de datos puntero.[6]

[6]La ventaja de la definición del tipo simbólico respecto a la utilización del tipo explícito es

Al igual que C, el lenguaje C++ posee operadores para asignación (`=`) y comparación (`==` y `!=`) entre punteros del mismo tipo. Del mismo modo, también posee dos operadores de desreferenciación: el operador prefijo *asterisco* (`*`) y el operador postfijo *flecha* (`->`).

```
typedef struct Nodo* PNodo ;
struct Nodo {
    PNodo sig ;
    int   dato ;
} ;
int main()
{
    PNodo lista = NULL ;
    /* ... */
    if (lista != NULL) {
        lista->sig = NULL ;
        lista->dato = 5 ;
    }
    /* ... */
}
```

8.2.2. La gestión de memoria dinámica en C++

Aunque los mecanismos de gestión de la memoria dinámica en C (véase 8.1.2) están soportados en C++, no resultan adecuados en C++. Ello es así porque C++ está fuertemente basado en la invocación automática de constructores y destructores, mientras que los mecanismos proporcionados en C no ofrecen dicho soporte.

Así, C++ define dos nuevos operadores (`new` y `delete`) que permiten reservar y liberar memoria dinámica, pero invocando a los contructores y destructores de los objetos que se crean y destruyen respectivamente.

- `ptr = new Tipo`: reserva un área de memoria dinámica de tamaño adecuado para contener un objeto de tipo `Tipo`, y lo construye invocando al constructor especificado. Finalmente devuelve un puntero al objeto alojado y construido en memoria dinámica.
- `delete ptr`: destruye, invocando a su destructor, el objeto apuntado por `ptr`, y libera el área de memoria dinámica que ocupaba. Este objeto debe haber sido construido utilizando el operador `new`.

```
typedef struct Dato* PDato ;
struct Dato {
    int valor ;
} ;
int main()
{
    PDato ptr = new Dato ;
    ptr->valor = 5 ;
    /* ... */
    delete ptr ;
}
```

C++ también proporciona mecanismos para alojar y desalojar un array de elementos en memoria dinámica, sin embargo, son raramente utilizados, ya que C++ proporciona el tipo `vector` en la biblioteca estándar, que proporciona una abstracción de alto nivel más flexible y potente.

```
typedef struct Dato* PDato ;
struct Dato {
    int valor ;
```

equivalente a la ventaja de la definición de constantes simbólicas respecto a la utilización de constantes explícitas. Además, favorece la utilización de punteros inteligentes (véase 8.2.7).

```
} ;
int main()
{
    PDato ptr = new Dato[5] ;
    for (int i = 0; i < 5; ++i) {
        ptr[i].valor = 7 ;
    }
    /* ... */
    delete [] ptr ;
}
```

8.2.3. El operador de dirección en C++

El operador de dirección es similar tanto en C como en C++ (véase 8.1.3). Sin embargo, es bastante menos utilizado en C++, ya que éste dispone del paso de parámetros por referencia. No obstante, suele ser utilizado a menudo para evitar la autoasignación en la implementación del operador de asignación:

```
Dato& operator= (const Dato& o)
{
    if (this != &o) {
        /* asignación */
    }
    return *this;
}
```

8.2.4. Arrays y aritmética de punteros en C++

Aunque C++ soporta la aritmética de punteros y la conversión automática de arrays predefinidos (véase 8.1.4), dichas características no suelen ser utilizadas con frecuencia. Ello es así porque C++ está más orientado a crear abstracciones de alto nivel sobre los datos. Dichas características de bajo nivel sólo son utilizadas en contadas ocasiones y siempre dentro de abstracciones que encapsulen estos mecanismos de bajo nivel. De hecho, el tipo `vector` de la biblioteca estándar proporciona una abstracción adecuada de alto nivel.

8.2.5. Punteros genéricos en C++

C++ soporta la posibilidad de trabajar con punteros a tipos desconocidos (genéricos de bajo nivel) (`void*`). Sin embargo, no suelen ser utilizados, salvo en contadas ocasiones que requieran programación de bajo nivel, porque pueden dar lugar a construcciones erróneas. De hecho, a diferencia de C (véase 8.1.5), en C++ las conversiones entre `void*` y otros tipos de punteros **no** son automáticas, y requieren conversiones explícitas.

Además, C++ proporciona un mecanismo más adecuado y robusto (con comprobación estricta de tipos) para soportar la genericidad mediante las plantillas (*templates*).

8.2.6. Tipos opacos en C++

Al igual que C (véase 8.1.6), C++ permite la definición de punteros a tipos incompletos (registros y clases), los cuales podrán ser definidos en módulos de implementación independientes. Estos tipos incompletos deberán ser definidos en el módulo de implementación antes de poder ser desreferenciados.

Además, la utilización de punteros a tipos incompletos dentro de clases permite definir TADs con implementación oculta.

```
//- dato.hpp ----------------------------
#ifndef dato_hpp_
#define dato_hpp_
class Dato {
    struct DatoImpl ;
    struct DatoImpl* dato ;
public:
    Dato() ;
    ~Dato() ;
    void incrementar(int x) ;
} ;
#endif
//- dato.cpp ----------------------------
#include "dato.hpp"
struct Dato::DatoImpl {
    int valor ;
};
Dato::Dato() : dato()
{
    dato = new DatoImpl ;
    dato->valor = 0 ;
}
void Dato::incrementar(int x)
{
    dato->valor += x ;
}
Dato::~Dato()
{
    delete dato ;
}
//- main.cpp ----------------------------
#include "dato.hpp"
int main()
{
    Dato dat ;
    dat.incrementar(5) ;
}
//---------------------------------------
```

8.2.7. Punteros inteligentes en C++

La potencia de C++, con contructores, destructores y sobrecarga de operadores, permite definir punteros inteligentes que gestionen automáticamente la liberación de los recursos (memoria dinámica) asociados a ellos.

La biblioteca estándar (C++11) define los siguientes punteros inteligentes: `shared_ptr`, `weak_ptr` y `unique_ptr`, así como el obsoleto `auto_ptr`. Para compiladores anteriores al estándar de 2011, se debe utilizar la biblioteca TR1.

Así mismo, tambien hace posible la definición de punteros inteligentes que proporcionen una considerable ayuda en la depuración de programas y en el aprendizaje de los conceptos y practica de punteros y gestión de memoria dinámica.

9. Conclusión

Este documento presenta las primitivas proporcionadas por los lenguajes de programación C y C++ como herramientas docentes para el aprendizaje de la programación desde un ámbito académico.

Así, se ha podido mostrar como para cada construcción alternativa que presenta C++ respecto a C, la alternativa presentada por C++ proporciona un esquema más simple, seguro y robusto, ante la introducción de posibles errores actuales y futuros, respecto a la propuesta original proporcionada por C.

A. Problemas del uso de `printf` y `scanf`: un ejemplo

A continuación mostramos un ejemplo donde se puede comprobar cómo el uso de `scanf` para la entrada de datos hace que un mismo programa produzca error o no dependiendo de la máquina para la que se compila.

Nota: para solventar el problema indicado (basado en el estándar ANSI C89), en el estándar C99 existe el *especificador* `z` para el tipo `size_t`.

```
#include <stdio.h>

struct Dato_1 {
    int x ;
    size_t y ;
    int z ;
} ;

struct Dato_2 {
    int x ;
    unsigned int y ;
    int z ;
} ;

struct Dato_3 {
    int x ;
    unsigned long long y ;
    int z ;
} ;

/*
 * El tipo size_t es el tipo del valor devuelto por el operador
 * sizeof(...), así como el tipo de numerosos parámetros de la
 * biblioteca estándar de C (y C++).
 *
 * Aunque el tipo size_t no es el centro de la argumentación
 * que se describe en este ejemplo, si da idea de los problemas
 * asociados al modelo de entrada/salida de C, donde una cadena
 * de formato controla la entrada/salida.
 *
 * El principal problema con size_t es que es un tipo definido
 * por la biblioteca estándar sobre el que no se especifica su
 * tipo base, y por lo tanto en algunos sistemas de 32 bits se
 * define como un tipo unsigned int (de 32 bits), sin embargo,
 * en algunos sistemas de 64 bits se define como un tipo
 * unsigned long (de 64 bits). Por lo que hacer código correcto
 * que compile adecuadamente en cualquier máquina, independientemente
 * de si es de 32 o 64 bits se vuelve una tarea complicada sin
 * utilizar directivas de compilación condicional (también
 * dependientes de la máquina).
 *
 * Este tipo de problema también surge cuando se cambia el tipo
 * de una variable, parámetro o campo de estructura, por ejemplo
 * de float a double, de unsigned a int, de short a int, de int
 * a long, etc.
 */


void p0_32_bits()
{
    /*
     * En una máquina de 32 bits, este código debe compilar
     * adecuadamente, ya que size_t es de tipo unsigned int.
     * Sin embargo, en una máquina de 64 bits este código
     * no compilará adecuadamente (GCC -Wall), y producirá
     * código ejecutable erróneo (en caso de compilar sin flags)
     */
    printf("sizeof(char): %u\n", sizeof(char)) ;
}

void p0_64_bits()
```

```
{
    /*
     * En una máquina de 64 bits, este código debe compilar
     * adecuadamente, ya que size_t es de tipo unsigned long.
     * Sin embargo, en una máquina de 32 bits este código
     * no compilará adecuadamente (GCC -Wall), y producirá
     * código ejecutable erróneo (en caso de compilar sin flags)
     */
    printf("sizeof(char): %lu\n", sizeof(char)) ;
}

void p1_32_bits()
{
    /*
     * Este ejemplo muestra cómo sería la Entrada/Salida de
     * un elemento de tipo size_t en una máquina de 32 bits.
     * En una máquina de 64 bits produciría errores de compilación
     * (GCC -Wall) o código erróneo que afectaría a los valores
     * de los otros campos.
     */
    struct Dato_1 d = { 5, 7, 9 } ;

    printf(" X: %d Y: %u Z: %d\n", d.x, d.y, d.z) ;
    printf("Introduce un numero: ") ;
    scanf(" %u", &d.y) ;
    printf(" X: %d Y: %u Z: %d\n", d.x, d.y, d.z) ;
}

void p1_64_bits()
{
    /*
     * Este ejemplo muestra cómo sería la Entrada/Salida de
     * un elemento de tipo size_t en una máquina de 64 bits.
     * En una máquina de 32 bits produciría errores de compilación
     * (GCC -Wall) o código erróneo que afectaría a los valores
     * de los otros campos.
     */
    struct Dato_1 d = { 5, 7, 9 } ;
    printf(" X: %d Y: %lu Z: %d\n", d.x, d.y, d.z) ;
    printf("Introduce un numero: ") ;
    scanf(" %lu", &d.y) ;
    printf(" X: %d Y: %lu Z: %d\n", d.x, d.y, d.z) ;
}

void p2()
{
    /*
     * Ejemplo para mostrar los problemas de código donde el
     * formato de Entrada/Salida especifica 64 bits, y el tipo
     * de datos es de 32 bits. Se puede apreciar cómo la salida
     * es incorrecta, y la entrada de datos modifica otras
     * variables (Z).
     */
    struct Dato_2 d = { 5, 7, 9 } ;
    printf("Valores correctos: X: %d Y: %u Z: %d\n", d.x, d.y, d.z) ;
    printf(" X: %d Y: %llu Z: %d\n", d.x, d.y, d.z) ;
    printf("Introduce un numero: ") ;
    scanf(" %llu", &d.y) ;
    printf(" X: %d Y: %llu Z: %d\n", d.x, d.y, d.z) ;
    printf("Valores correctos: X: %d Y: %u Z: %d\n", d.x, d.y, d.z) ;
}

void p3()
{
    /*
     * Ejemplo para mostrar los problemas de código donde el
     * formato de Entrada/Salida especifica 32 bits, y el tipo
     * de datos es de 64 bits. Se puede apreciar cómo la salida
     * es incorrecta. El error en la entrada de datos dependerá
     * de si la máquina es big-endian o little-endian.
     */
    struct Dato_3 d = { 5, 7, 9 } ;
    printf("Valores correctos: X: %d Y: %llu Z: %d\n", d.x, d.y, d.z) ;
    printf(" X: %d Y: %u Z: %d\n", d.x, d.y, d.z) ;
```

```
    printf("Introduce un numero: ") ;
    scanf(" %u", &d.y) ;
    printf(" X: %d Y: %u Z: %d\n", d.x, d.y, d.z) ;
    printf("Valores correctos: X: %d Y: %llu Z: %d\n", d.x, d.y, d.z) ;
}

int main()
{
    printf("--------------------------------\n") ;
    p0_32_bits() ;
    printf("--------------------------------\n") ;
    p0_64_bits() ;
    printf("--------------------------------\n") ;
    p1_32_bits() ;
    printf("--------------------------------\n") ;
    p1_64_bits() ;
    printf("--------------------------------\n") ;
    p2() ;
    printf("--------------------------------\n") ;
    p3() ;
    printf("--------------------------------\n") ;
    return 0 ;
}
```